Leveraging a functional 🤖 approach for easier testing and maintenance of ROS 2 code

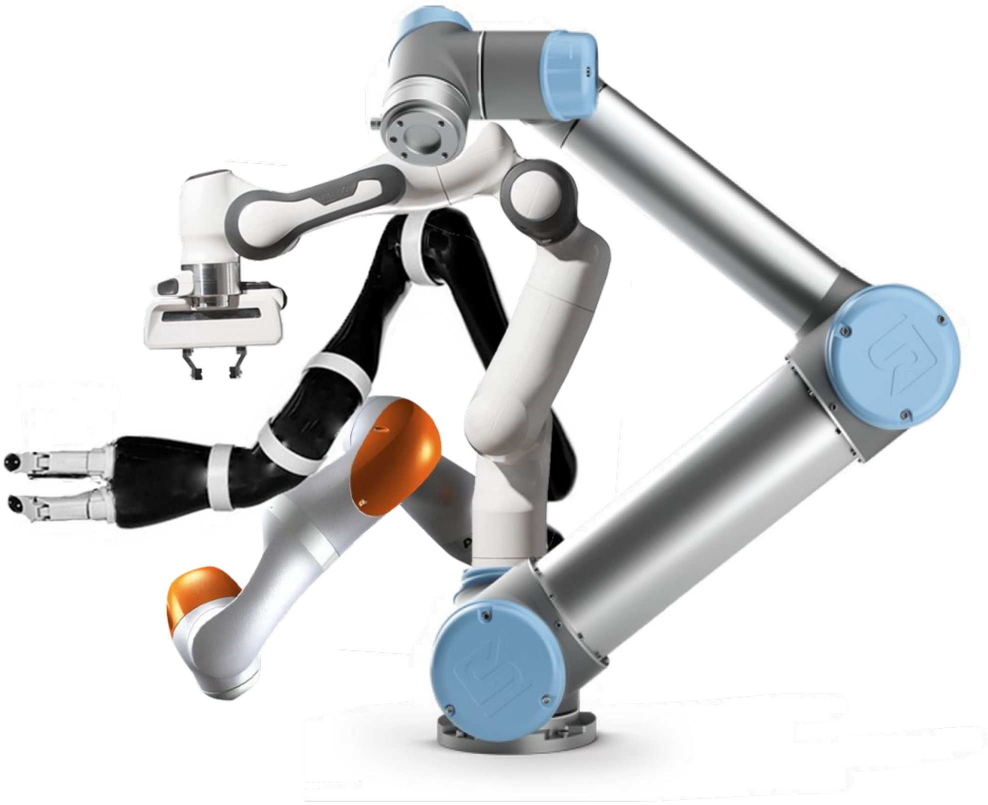
October 2023

Bilal Gill
bilal.gill@picknik.ai

PICKNIK

Outline

- Introduction
- ROS 2 Conventional Approach
- Introduction to Functional Programming Principles
- Refactoring using Functional Programming Principles
- Conclusion



Introduction

About Me

- Robotics Engineer on the services team at PickNik Robotics
  - Contributed to a wide variety of client projects: remotely operated underwater inspection vehicles, autonomous mobile base for agriculture applications, and more
- Have worked at General Dynamics Electric Boat, MIT Lincoln Laboratory
- Interested in robotics since high school

PICKNIK

Why give this talk?

- Engineers at PickNik experiment with different ways to architect code that uses ROS 2 – creating code that is easy to understand, test, maintain, and extend

Why give this talk?

- Engineers at PickNik experiment with different ways to architect code that uses ROS 2 – creating code that is easy to understand, test, maintain, and extend
- At the end of a services project, code is handed over to the client

Why give this talk?

- Engineers at PickNik experiment with different ways to architect code that uses ROS 2 – creating code that is easy to understand, test, maintain, and extend
- At the end of a services project, code is handed over to the client
- How can the client expect proper operation of the software once they start developing on top of it?

Why give this talk?

- Engineers at PickNik experiment with different ways to architect code that uses ROS 2 – creating code that is easy to understand, test, maintain, and extend
- At the end of a services project, code is handed over to the client
- How can the client expect proper operation of the software once they start developing on top of it?

Answer: Tests and documentation! ***Lots and lots of documentation!***

PICKNIK

Why give this talk?

- Engineers at PickNik experiment with different ways to architect code that uses ROS 2 – creating code that is easy to understand, test, maintain, and extend
- At the end of a services project, code is handed over to the client
- How can the client expect proper operation of the software once they start developing on top of it?

Answer: Tests and documentation! ***Lots and lots of documentation!***

- The tests written for the software **should not be flaky!**
  - Flaky tests: tests that return both passes and failures despite no changes to the code or the test itself

PICKNIK

Why give this talk?

- Engineers at PickNik experiment with different ways to architect code that uses ROS 2 – creating code that is easy to understand, test, maintain, and extend
- At the end of a services project, code is handed over to the client
- How can the client expect proper operation of the software once they start developing on top of it?

Answer: Tests and documentation! ***Lots and lots of documentation!***

- The tests written for the software **should not be flaky!**
  - Flaky tests: tests that return both passes and failures despite no changes to the code or the test itself
- ROS 2 documentation encourages an object-oriented paradigm that can lead to trouble writing code that achieves the goal

PICKNIK

Why give this talk?

- Engineers at PickNik experiment with different ways to architect code that uses ROS 2 – creating code that is easy to understand, test, maintain, and extend
- At the end of a services project, code is handed over to the client
- How can the client expect proper operation of the software once they start developing on top of it?

Answer: Tests and documentation! ***Lots and lots of documentation!***

- The tests written for the software **should not be flaky!**
  - Flaky tests: tests that return both passes and failures despite no changes to the code or the test itself
- ROS 2 documentation encourages an object-oriented paradigm that can lead to trouble writing code that achieves the goal
- **Adopting functional programming techniques into our code has made it easier to test, maintain, and extend code!**

Motivating Example

- Problem: A robot wants to navigate from its current location to some goal

Clip from https://www.youtube.com/watch?v=VTeY-l-Xh6c

Motivating Example

- Problem: A robot wants to navigate from its current location to some goal
- The robot needs to know where obstacles are located in its environment

Clip from https://www.youtube.com/watch?v=VTeY-l-Xh6c

PICKNIK

Motivating Example

- Problem: A robot wants to navigate from its current location to some goal
- The robot needs to know where obstacles are located in its environment
- Let's use an occupancy map to represent the environment

Clip from https://www.youtube.com/watch?v=VTeY-l-Xh6c

PICKNIK

Motivating Example

- Problem: A robot wants to navigate from its current location to some goal
- The robot needs to know where obstacles are located in its environment
- Let's use an occupancy map to represent the environment
- Assumption: The robot knows its location in the occupancy map at all times

Clip from https://www.youtube.com/watch?v=VTeY-l-Xh6c

Motivating Example

- Problem: A robot wants to navigate from its current location to some goal
- The robot needs to know where obstacles are located in its environment
- Let’s use an occupancy map to represent the environment
- Assumption: The robot knows its location in the occupancy map at all times
- Solution: The robot will send a request to a ROS 2 service that generates a path from the robot’s current location and goal location, given an occupancy map

Clip from https://www.youtube.com/watch?v=VTeY-l-Xh6c

PICKNIK

ROS 2 Conventional Approach

Conventional Approach

```
class PathGenerator : public rclcpp::Node {
 public:
  explicit PathGenerator(
          rclcpp::NodeOptions const& options = rclcpp::NodeOptions{})
           : Node("path_generator", options);

 private:
  void set_map_service(
      const std::shared_ptr<SetMap::Request> request,
      std::shared_ptr<SetMap::Response> response);

  void generate_path_service(
      const std::shared_ptr<GetPath::Request> request,
      std::shared_ptr<GetPath::Response> response);

  bool set_costmap(const std_msgs::msg::UInt8MultiArray& costmap);

  Path generate_global_path(Position const& start,Position const& goal);

  Map<unsigned char> map_;
  int robot_size_;
  std::unique_ptr<CollisionChecker<unsigned char>> is_occupied_;
  rclcpp::Service<SetMap>::SharedPtr map_setter_service_;
  rclcpp::Service<GetPath>::SharedPtr path_generator_service_;
};
```

- `PathGenerator` will be used to generate the path for our robot
- This code was written using example code available from the ROS 2 documentation
- Testing this class requires spinning up clients to send requests to the services and inspecting the responses

PICKNIK

Conventional Approach

```
class PathGenerator : public rclcpp::Node {
 public:
  explicit PathGenerator(
          rclcpp::NodeOptions const& options = rclcpp::NodeOptions{})
           : Node("path_generator", options);

 private:
  void set_map_service(
       const std::shared_ptr<SetMap::Request> request,
       std::shared_ptr<SetMap::Response> response);

  void generate_path_service(
       const std::shared_ptr<GetPath::Request> request,
       std::shared_ptr<GetPath::Response> response);

  bool set_costmap(const std_msgs::msg::UInt8MultiArray& costmap);

  Path generate_global_path(Position const& start,Position const& goal);

  Map<unsigned char> map_;
  int robot_size_;
  std::unique_ptr<CollisionChecker<unsigned char>> is_occupied_;
  rclcpp::Service<SetMap>::SharedPtr map_setter_service_;
  rclcpp::Service<GetPath>::SharedPtr path_generator_service_;
};
```

- PathGenerator will be used to generate the path for our robot
- This code was written using example code available from the ROS 2 documentation
- Testing this class requires spinning up clients to send requests to the services and inspecting the responses
- **Involving inter-process communication in unit tests introduces flakiness**

PICKNIK

Conventional Approach

```
class PathGenerator : public rclcpp::Node {
 public:
  explicit PathGenerator(
          rclcpp::NodeOptions const& options = rclcpp::NodeOptions{})
           : Node("path_generator", options);

 private:
  void set_map_service(
       const std::shared_ptr<SetMap::Request> request,
       std::shared_ptr<SetMap::Response> response);

  void generate_path_service(
       const std::shared_ptr<GetPath::Request> request,
       std::shared_ptr<GetPath::Response> response);

  bool set_costmap(const std_msgs::msg::UInt8MultiArray& costmap);

  Path generate_global_path(Position const& start,Position const& goal);

  Map<unsigned char> map_;
  int robot_size_;
  std::unique_ptr<CollisionChecker<unsigned char>> is_occupied_;
  rclcpp::Service<SetMap>::SharedPtr map_setter_service_;
  rclcpp::Service<GetPath>::SharedPtr path_generator_service_;
};
```

- `PathGenerator` will be used to generate the path for our robot
- This code was written using example code available from the ROS 2 documentation
- Testing this class requires spinning up clients to send requests to the services and inspecting the responses
- **Involving inter-process communication in unit tests introduces flakiness**
- Is there a way to refactor this code such that invoking the ROS 2 API is avoided?

PICKNIK

Conventional Approach

```cpp
class PathGenerator : public rclcpp::Node {
 public:
  explicit PathGenerator(
          rclcpp::NodeOptions const& options = rclcpp::NodeOptions{})
           : Node("path_generator", options);

 private:
  void set_map_service(
       const std::shared_ptr<SetMap::Request> request,
       std::shared_ptr<SetMap::Response> response);

  void generate_path_service(
       const std::shared_ptr<GetPath::Request> request,
       std::shared_ptr<GetPath::Response> response);

  bool set_costmap(const std_msgs::msg::UInt8MultiArray& costmap);

  Path generate_global_path(Position const& start,Position const& goal);

  Map<unsigned char> map_;
  int robot_size_;
  std::unique_ptr<CollisionChecker<unsigned char>> is_occupied_;
  rclcpp::Service<SetMap>::SharedPtr map_setter_service_;
  rclcpp::Service<GetPath>::SharedPtr path_generator_service_;
};
```

- `PathGenerator` will be used to generate the path for our robot
- This code was written using example code available from the ROS 2 documentation
- Testing this class requires spinning up clients to send requests to the services and inspecting the responses
- **Involving inter-process communication in unit tests introduces flakiness**
- Is there a way to refactor this code such that invoking the ROS 2 API is avoided?
- Yes, by using functional programming principles

PICKNIK

Introduction to Functional Programming Principles

PICKNIK

What is Functional Programming?

- A programming paradigm characterized by the use of pure functions and the avoidance of side effects

PICKNIK

What is Functional Programming?

- A programming paradigm characterized by the use of pure functions and the avoidance of side effects
- Functional programming is identified by the use of higher order functions, pure functions, monads, declarative syntax

PICKNIK

What is Functional Programming?

- A programming paradigm characterized by the use of pure functions and the avoidance of side effects
- Functional programming is identified by the use of higher order functions, pure functions, monads, declarative syntax
  - C++ has all the tools to implement functional programming, including lambda functions, std::function, std::optional, std::expected, and more

PICKNIK

What is Functional Programming?

- A programming paradigm characterized by the use of pure functions and the avoidance of side effects
- Functional programming is identified by the use of higher order functions, pure functions, monads, declarative syntax
    - C++ has all the tools to implement functional programming, including lambda functions, std::function, std::optional, std::expected, and more
    - Want to maximize use of these features to write code with a minimal number of side effects

PICKNIK

What is Functional Programming?

- A programming paradigm characterized by the use of pure functions and the avoidance of side effects
- Functional programming is identified by the use of higher order functions, pure functions, monads, declarative syntax
  - C++ has all the tools to implement functional programming, including lambda functions, std::function, std::optional, std::expected, and more
  - Want to maximize use of these features to write code with a minimal number of side effects
- Let's go over some principles and see how we can use them in refactoring `PathGenerator`

PICKNIK

Pure Functions

- A pure function is a function that
    - is deterministic: They always return the same output for the same set of inputs

Pure Functions

- A pure function is a function that
  - is deterministic: They always return the same output for the same set of inputs
  - has no side effects: They don't alter the state of the program, global variables, or produce any observable interactions with the outside world, such as writing to files or displaying output

Pure Functions

- A pure function is a function that
  - is deterministic: They always return the same output for the same set of inputs
  - has no side effects: They don't alter the state of the program, global variables, or produce any observable interactions with the outside world, such as writing to files or displaying output
- Why is this desirable?

Pure Functions

- A pure function is a function that
  - is deterministic: They always return the same output for the same set of inputs
  - has no side effects: They don't alter the state of the program, global variables, or produce any observable interactions with the outside world, such as writing to files or displaying output
- Why is this desirable?
  - local reasoning: The code can be understood just by looking at that portion and a limited scope around it

Pure Functions

- A pure function is a function that
  - is deterministic: They always return the same output for the same set of inputs
  - has no side effects: They don't alter the state of the program, global variables, or produce any observable interactions with the outside world, such as writing to files or displaying output
- Why is this desirable?
  - local reasoning: The code can be understood just by looking at that portion and a limited scope around it
  - testability: Testing pure functions is trivial

Pure Functions

- A pure function is a function that
  - is deterministic: They always return the same output for the same set of inputs
  - has no side effects: They don’t alter the state of the program, global variables, or produce any observable interactions with the outside world, such as writing to files or displaying output
- Why is this desirable?
  - local reasoning: The code can be understood just by looking at that portion and a limited scope around it
  - testability: Testing pure functions is trivial
- Practically, pure functions do not:
  - contain state (static variables or class member variables)

PICKNIK

Pure Functions

- A pure function is a function that
  - is deterministic: They always return the same output for the same set of inputs
  - has no side effects: They don't alter the state of the program, global variables, or produce any observable interactions with the outside world, such as writing to files or displaying output
- Why is this desirable?
  - local reasoning: The code can be understood just by looking at that portion and a limited scope around it
  - testability: Testing pure functions is trivial
- Practically, pure functions do not:
  - contain state (static variables or class member variables)
  - mutate input parameters

Pure Functions

- A pure function is a function that
  - is deterministic: They always return the same output for the same set of inputs
  - has no side effects: They don't alter the state of the program, global variables, or produce any observable interactions with the outside world, such as writing to files or displaying output
- Why is this desirable?
  - local reasoning: The code can be understood just by looking at that portion and a limited scope around it
  - testability: Testing pure functions is trivial
- Practically, pure functions do not:
  - contain state (static variables or class member variables)
  - mutate input parameters
- A function is pure if and only if it could be replaced by a lookup table (potentially infinitely large!)

PICKNIK

Higher Order Functions

- A higher order function is a function that can:
  - accept other functions as arguments

Higher Order Functions

- A higher order function is a function that can:
  - accept other functions as arguments
  - return functions as a result

PICKNIK

Higher Order Functions

- A higher order function is a function that can:
  - accept other functions as arguments
  - return functions as a result
- Why is this desirable?

Higher Order Functions

- A higher order function is a function that can:
  - accept other functions as arguments
  - return functions as a result
- Why is this desirable?
  - Modularity and reusability: The behavior of a higher order function is configurable and they can be used in different contexts

Higher Order Functions

- A higher order function is a function that can:
  - accept other functions as arguments
  - return functions as a result
- Why is this desirable?
  - Modularity and reusability: The behavior of a higher order function is configurable and they can be used in different contexts
  - Control flow abstraction: control flows, like looping and conditional execution, can be abstracted in a readable and reusable manner

PICKNIK

Higher Order Functions

- A higher order function is a function that can:
  - accept other functions as arguments
  - return functions as a result
- Why is this desirable?
  - Modularity and reusability: The behavior of a higher order function is configurable and they can be used in different contexts
  - Control flow abstraction: control flows, like looping and conditional execution, can be abstracted in a readable and reusable manner
  - Testability: Smaller well-defined functions are easier to test

PICKNIK

Higher Order Functions

- A higher order function is a function that can:
  - accept other functions as arguments
  - return functions as a result
- Why is this desirable?
  - Modularity and reusability: The behavior of a higher order function is configurable and they can be used in different contexts
  - Control flow abstraction: control flows, like looping and conditional execution, can be abstracted in a readable and reusable manner
  - Testability: Smaller well-defined functions are easier to test
- The Standard Template Library contains many higher order functions!

Higher Order Functions

- A higher order function is a function that can:
  - accept other functions as arguments
  - return functions as a result
- Why is this desirable?
  - Modularity and reusability: The behavior of a higher order function is configurable and they can be used in different contexts
  - Control flow abstraction: control flows, like looping and conditional execution, can be abstracted in a readable and reusable manner
  - Testability: Smaller well-defined functions are easier to test
- The Standard Template Library contains many higher order functions!
  - std::transform, std::find_if, std::copy, and more

PICKNIK

Monadic Error Handling

- Monadic error handling is a functional programming technique for dealing with errors in a clean, compositional, and type-safe way

Monadic Error Handling

- Monadic error handling is a functional programming technique for dealing with errors in a clean, compositional, and type-safe way
- This approach encapsulates error handling into types and operations on these types

Monadic Error Handling

- Monadic error handling is a functional programming technique for dealing with errors in a clean, compositional, and type-safe way
- This approach encapsulates error handling into types and operations on these types
- Why is this desirable?

PICKNIK

Monadic Error Handling

- Monadic error handling is a functional programming technique for dealing with errors in a clean, compositional, and type-safe way
- This approach encapsulates error handling into types and operations on these types
- Why is this desirable?
    - Type encapsulation: Monadic error handling encapsulates the result of computations along with possible errors within a single type

PICKNIK

Monadic Error Handling

- Monadic error handling is a functional programming technique for dealing with errors in a clean, compositional, and type-safe way
- This approach encapsulates error handling into types and operations on these types
- Why is this desirable?
    - Type encapsulation: Monadic error handling encapsulates the result of computations along with possible errors within a single type
    - Compositional error handling: Monadic error handling allows composition of operations that might fail, in a way that if any operation fails, the whole computation fails

PICKNIK

Monadic Error Handling

- Monadic error handling is a functional programming technique for dealing with errors in a clean, compositional, and type-safe way
- This approach encapsulates error handling into types and operations on these types
- Why is this desirable?
    - Type encapsulation: Monadic error handling encapsulates the result of computations along with possible errors within a single type
    - Compositional error handling: Monadic error handling allows composition of operations that might fail, in a way that if any operation fails, the whole computation fails
    - Error Propagation: Errors can be automatically propagated through a sequence of computations until they are explicitly handled

How does functional programming help?

- Functional programming lends itself to the minimization of mutable state

PICKNIK

How does functional programming help?

- Functional programming lends itself to the minimization of mutable state
- Testing is easier with pure functions because only the return value of the function needs to be evaluated

How does functional programming help?

- Functional programming lends itself to the minimization of mutable state
- Testing is easier with pure functions because only the return value of the function needs to be evaluated
- Different functions can be passed as arguments to higher order functions, lending itself to modularity

How does functional programming help?

- Functional programming lends itself to the minimization of mutable state
- Testing is easier with pure functions because only the return value of the function needs to be evaluated
- Different functions can be passed as arguments to higher order functions, lending itself to modularity
- Monadic error handling simplifies error checking

How does functional programming help?

- Functional programming lends itself to the minimization of mutable state
- Testing is easier with pure functions because only the return value of the function needs to be evaluated
- Different functions can be passed as arguments to higher order functions, lending itself to modularity
- Monadic error handling simplifies error checking
- Let’s try and refactor `PathGenerator`

PICKNIK

How does functional programming help?

- Functional programming lends itself to the minimization of mutable state
- Testing is easier with pure functions because only the return value of the function needs to be evaluated
- Different functions can be passed as arguments to higher order functions, lending itself to modularity
- Monadic error handling simplifies error checking
- Let's try and refactor `PathGenerator`
    - **Claim: that the refactored `PathGenerator` has 100% coverage**

Refactoring using Functional Programming Principles

PICKNIK

Refactoring PathGenerator

```cpp
class PathGenerator : public rclcpp::Node {
 public:
  explicit PathGenerator(rclcpp::NodeOptions const&
                         options = rclcpp::NodeOptions{})
           : Node("path_generator", options);

 private:
  void set_map_service(
     const std::shared_ptr<SetMap::Request> request,
     std::shared_ptr<SetMap::Response> response);

  void generate_path_service(
     const std::shared_ptr<GetPath::Request> request,
     std::shared_ptr<GetPath::Response> response);

  bool set_costmap(const std_msgs::msg::UInt8MultiArray& costmap);

  Path generate_global_path(Position const& start,Position const& goal);

  /* Additional private members*/
};
```

- How the current `PathGenerator` looks

Refactoring PathGenerator

```
class PathGenerator : public rclcpp::Node {
 public:
  explicit PathGenerator(rclcpp::NodeOptions const&
                         options = rclcpp::NodeOptions{})
          : Node("path_generator", options);

 private:
  void set_map_service(
     const std::shared_ptr<SetMap::Request> request,
     std::shared_ptr<SetMap::Response> response);

  void generate_path_service(
     const std::shared_ptr<GetPath::Request> request,
     std::shared_ptr<GetPath::Response> response);

  bool set_costmap(const std_msgs::msg::UInt8MultiArray& costmap);

  Path generate_global_path(Position const& start,Position const& goal);

  /* Additional private members*/
};
```

- A `rclcpp::Node` object can be constructed in `main` and services can be assigned
- No need to have a `PathGenerator` object that inherits from `rclcpp::Node`

PICKNIK

Refactoring PathGenerator

```
class PathGenerator : public rclcpp::Node {
 public:
  explicit PathGenerator(rclcpp::NodeOptions const&
                         options = rclcpp::NodeOptions{})
          : Node("path_generator", options);

 private:
  void set_map_service(
     const std::shared_ptr<SetMap::Request> request,
     std::shared_ptr<SetMap::Response> response);

  void generate_path_service(
     const std::shared_ptr<GetPath::Request> request,
     std::shared_ptr<GetPath::Response> response);

  bool set_costmap(const std_msgs::msg::UInt8MultiArray& costmap);

  Path generate_global_path(Position const& start,Position const& goal);

  /* Additional private members*/
};
```

- A `rclcpp::Node` object can be constructed in `main` and services can be assigned
- No need to have a `PathGenerator` object that inherits from `rclcpp::Node`
- The `create_service` method accepts class methods, free functions, and lambdas as the callback function
- The private functions of `PathGenerator` can be turned into free functions and lambda functions

PICKNIK

Refactoring PathGenerator

```
class PathGenerator : public rclcpp::Node {
 public:
  explicit PathGenerator(rclcpp::NodeOptions const&
                         options = rclcpp::NodeOptions{})
          : Node("path_generator", options);

 private:
  void set_map_service(
     const std::shared_ptr<SetMap::Request> request,
     std::shared_ptr<SetMap::Response> response);

  void generate_path_service(
     const std::shared_ptr<GetPath::Request> request,
     std::shared_ptr<GetPath::Response> response);

  bool set_costmap(const std_msgs::msg::UInt8MultiArray& costmap);

  Path generate_global_path(Position const& start,Position const& goal);

  /* Additional private members*/
};
```

- A `rclcpp::Node` object can be constructed in `main` and services can be assigned
- No need to have a `PathGenerator` object that inherits from `rclcpp::Node`
- The `create_service` method accepts class methods, free functions, and lambdas as the callback function
- The private functions of `PathGenerator` can be turned into free functions and lambda functions
- Let's refactor the callback function for the generate path service

PICKNIK

Refactoring PathGenerator

```
void generate_path_service(
const std::shared_ptr<GetPath::Request> request,
      std::shared_ptr<GetPath::Response> response) {
  if (map_.get_data().size() == 0) {
    RCLCPP_ERROR_STREAM(this->get_logger(), "MAP IS EMPTY!!");
    response->code.code = example_srvs::msg::GetPathCodes::EMPTY_OCCUPANCY_MAP;
    response->path = std_msgs::msg::UInt8MultiArray();
    return;
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path
  auto const path = generate_global_path(start, goal);

  // Start populating the response message
  auto response_path = std_msgs::msg::UInt8MultiArray();

  /* Code about populating the message here */

  response->code.code = !path.empty() ? example_srvs::msg::GetPathCodes::SUCCESS :
example_srvs::msg::GetPathCodes::NO_VALID_PATH;
    response->path = response_path;
  }
```


PICKNIK

Refactoring PathGenerator

```
void generate_path_service(
const std::shared_ptr<GetPath::Request> request,
      std::shared_ptr<GetPath::Response> response) {
  if (map_.get_data().size() == 0) {
    RCLCPP_ERROR_STREAM(this->get_logger(), "MAP IS EMPTY!!");
    response->code.code = example_srvs::msg::GetPathCodes::EMPTY_OCCUPANCY_MAP;
    response->path = std_msgs::msg::UInt8MultiArray();
    return;
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path
  auto const path = generate_global_path(start, goal);

  // Start populating the response message
  auto response_path = std_msgs::msg::UInt8MultiArray();

  /* Code about populating the message here */

  response->code.code = !path.empty() ? example_srvs::msg::GetPathCodes::SUCCESS :
example_srvs::msg::GetPathCodes::NO_VALID_PATH;
    response->path = response_path;
  }
```

- generate_path_service is:
  - printing errors

PICKNIK

Refactoring PathGenerator

```
void generate_path_service(
const std::shared_ptr<GetPath::Request> request,
      std::shared_ptr<GetPath::Response> response) {
  if (map_.get_data().size() == 0) {
    RCLCPP_ERROR_STREAM(this->get_logger(), "MAP IS EMPTY!!");
    response->code.code = example_srvs::msg::GetPathCodes::EMPTY_OCCUPANCY_MAP;
    response->path = std_msgs::msg::UInt8MultiArray();
    return;
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path
  auto const path = generate_global_path(start, goal);

  // Start populating the response message
  auto response_path = std_msgs::msg::UInt8MultiArray();

  /* Code about populating the message here */

  response->code.code = !path.empty() ? example_srvs::msg::GetPathCodes::SUCCESS :
example_srvs::msg::GetPathCodes::NO_VALID_PATH;
    response->path = response_path;
  }
```

- `generate_path_service` is:
  - printing errors
  - generating the path

Refactoring PathGenerator

```
void generate_path_service(
const std::shared_ptr<GetPath::Request> request,
      std::shared_ptr<GetPath::Response> response) {
  if (map_.get_data().size() == 0) {
    RCLCPP_ERROR_STREAM(this->get_logger(), "MAP IS EMPTY!!");
    response->code.code = example_srvs::msg::GetPathCodes::EMPTY_OCCUPANCY_MAP;
    response->path = std_msgs::msg::UInt8MultiArray();
    return;
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path
  auto const path = generate_global_path(start, goal);

  // Start populating the response message
  auto response_path = std_msgs::msg::UInt8MultiArray();

  /* Code about populating the message here */

  response->code.code = !path.empty() ? example_srvs::msg::GetPathCodes::SUCCESS :
example_srvs::msg::GetPathCodes::NO_VALID_PATH;
    response->path = response_path;
  }
```

- `generate_path_service` is:
  - printing errors
  - generating the path
  - setting an out parameter

PICKNIK

Refactoring PathGenerator

```
void generate_path_service(
const std::shared_ptr<GetPath::Request> request,
      std::shared_ptr<GetPath::Response> response) {
  if (map_.get_data().size() == 0) {
    RCLCPP_ERROR_STREAM(this->get_logger(), "MAP IS EMPTY!!");
    response->code.code = example_srvs::msg::GetPathCodes::EMPTY_OCCUPANCY_MAP;
    response->path = std_msgs::msg::UInt8MultiArray();
    return;
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path
  auto const path = generate_global_path(start, goal);

  // Start populating the response message
  auto response_path = std_msgs::msg::UInt8MultiArray();

  /* Code about populating the message here */

  response->code.code = !path.empty() ? example_srvs::msg::GetPathCodes::SUCCESS :
example_srvs::msg::GetPathCodes::NO_VALID_PATH;
    response->path = response_path;
  }
```

- `generate_path_service` is:
  - printing errors
  - generating the path
  - setting an out parameter
- Let’s isolate the error printing functionality to another function
  - The error printing function needs to be passed an error type

PICKNIK

Refactoring PathGenerator

```
void generate_path_service(
const std::shared_ptr<GetPath::Request> request,
      std::shared_ptr<GetPath::Response> response) {
  if (map_.get_data().size() == 0) {
    RCLCPP_ERROR_STREAM(this->get_logger(), "MAP IS EMPTY!!");
    response->code.code = example_srvs::msg::GetPathCodes::EMPTY_OCCUPANCY_MAP;
    response->path = std_msgs::msg::UInt8MultiArray();
    return;
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path
  auto const path = generate_global_path(start, goal);

  // Start populating the response message
  auto response_path = std_msgs::msg::UInt8MultiArray();

  /* Code about populating the message here */

  response->code.code = !path.empty() ? example_srvs::msg::GetPathCodes::SUCCESS :
example_srvs::msg::GetPathCodes::NO_VALID_PATH;
    response->path = response_path;
  }
```

- `generate_path_service` is:
  - printing errors
  - generating the path
  - setting an out parameter
- Let's isolate the error printing functionality to another function
  - The error printing function needs to be passed an error type
- The object held by the shared pointer can be assigned by another function

PICKNIK

Refactoring PathGenerator

```
void generate_path_service(
const std::shared_ptr<GetPath::Request> request,
      std::shared_ptr<GetPath::Response> response) {
  if (map_.get_data().size() == 0) {
    RCLCPP_ERROR_STREAM(this->get_logger(), "MAP IS EMPTY!!");
    response->code.code = example_srvs::msg::GetPathCodes::EMPTY_OCCUPANCY_MAP;
    response->path = std_msgs::msg::UInt8MultiArray();
    return;
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path
  auto const path = generate_global_path(start, goal);

  // Start populating the response message
  auto response_path = std_msgs::msg::UInt8MultiArray();

  /* Code about populating the message here */

  response->code.code = !path.empty() ? example_srvs::msg::GetPathCodes::SUCCESS :
example_srvs::msg::GetPathCodes::NO_VALID_PATH;
    response->path = response_path;
  }
```

- `generate_path_service` is:
  - printing errors
  - generating the path
  - setting an out parameter
- Let’s isolate the error printing functionality to another function
  - The error printing function needs to be passed an error type
- The object held by the shared pointer can be assigned by another function
- The `generate_global_path` function and associated pre-checks can be extracted to another function

PICKNIK

Refactoring PathGenerator

```
std::expected<GetPath::Response, error> generate_path(
  std::shared_ptr<GetPath::Request> const request,
  Map<unsigned char> const& occupancy_map, PathingGenerator path_generator) {
  if (occupancy_map.get_data().size() == 0) {
    return std::unexpected(error::EMPTY_OCCUPANCY_MAP);
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path using the path generator function that was input
  auto const path = path_generator(start, goal, occupancy_map);
  if (!path.has_value()) {
    return std::unexpected(error::NO_VALID_PATH);
  }

  auto response = GetPath::Response{};
  /* More implementation code */
  return response;
}
```

- Here is the refactored core functionality of the generate path callback

PICKNIK

Refactoring PathGenerator

```
std::expected<GetPath::Response, error> generate_path(
  std::shared_ptr<GetPath::Request> const request,
  Map<unsigned char> const& occupancy_map, PathingGenerator path_generator) {
  if (occupancy_map.get_data().size() == 0) {
    return std::unexpected(error::EMPTY_OCCUPANCY_MAP);
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path using the path generator function that was input
  auto const path = path_generator(start, goal, occupancy_map);
  if (!path.has_value()) {
    return std::unexpected(error::NO_VALID_PATH);
  }

  auto response = GetPath::Response{};
  /* More implementation code */
  return response;
}
```

- Here is the refactored core functionality of the generate path callback
- This function returns a type which can be used for **monadic error handling**

PICKNIK

Refactoring PathGenerator

```
std::expected<GetPath::Response, error> generate_path(
  std::shared_ptr<GetPath::Request> const request,
  Map<unsigned char> const& occupancy_map, PathingGenerator path_generator) {
  if (occupancy_map.get_data().size() == 0) {
    return std::unexpected(error::EMPTY_OCCUPANCY_MAP);
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path using the path generator function that was input
  auto const path = path_generator(start, goal, occupancy_map);
  if (!path.has_value()) {
    return std::unexpected(error::NO_VALID_PATH);
  }

  auto response = GetPath::Response{};
  /* More implementation code */
  return response;
}
```

- Here is the refactored core functionality of the generate path callback
- This function returns a type which can be used for **monadic error handling**
- If there is an error, the function can handle the error in a compile time checkable way

PICKNIK

Refactoring PathGenerator

```
using PathingGenerator = std::function<std::optional<Path>(
    Position const&, Position const&, Map<unsigned char> const&)>;

std::expected<GetPath::Response, error> generate_path(
  std::shared_ptr<GetPath::Request> const request,
  Map<unsigned char> const& occupancy_map, PathingGenerator path_generator) {
  if (occupancy_map.get_data().size() == 0) {
    return std::unexpected(error::EMPTY_OCCUPANCY_MAP);
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path using the path generator function that was input
  auto const path = path_generator(start, goal, occupancy_map);
  if (!path.has_value()) {
    return std::unexpected(error::NO_VALID_PATH);
  }

  auto response = GetPath::Response{};
  /* More implementation code */
  return response;
}
```

- Here is the refactored core functionality of the generate path callback
- This function returns a type which can be used for **monadic error handling**
- If there is an error, the function can handle the error in a compile time checkable way
- The function that generates the path can now be passed in, making this function a **higher order function**

PICKNIK

Refactoring PathGenerator

```
using PathingGenerator = std::function<std::optional<Path>(
    Position const&, Position const&, Map<unsigned char> const&)>;

std::expected<GetPath::Response, error> generate_path(
  std::shared_ptr<GetPath::Request> const request,
  Map<unsigned char> const& occupancy_map, PathingGenerator path_generator) {
  if (occupancy_map.get_data().size() == 0) {
    return std::unexpected(error::EMPTY_OCCUPANCY_MAP);
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path using the path generator function that was input
  auto const path = path_generator(start, goal, occupancy_map);
  if (!path.has_value()) {
    return std::unexpected(error::NO_VALID_PATH);
  }

  auto response = GetPath::Response{};
  /* More implementation code */
  return response;
}
```

- Here is the refactored core functionality of the generate path callback
- This function returns a type which can be used for **monadic error handling**
- If there is an error, the function can handle the error in a compile time checkable way
- The function that generates the path can now be passed in, making this function a **higher order function**
- This function is deterministic and has no side effects, so it is a **pure function**

PICKNIK

Refactoring PathGenerator

```
using PathingGenerator = std::function<std::optional<Path>(
    Position const&, Position const&, Map<unsigned char> const&)>;

std::expected<GetPath::Response, error> generate_path(
  std::shared_ptr<GetPath::Request> const request,
  Map<unsigned char> const& occupancy_map, PathingGenerator path_generator) {
  if (occupancy_map.get_data().size() == 0) {
    return std::unexpected(error::EMPTY_OCCUPANCY_MAP);
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path using the path generator function that was input
  auto const path = path_generator(start, goal, occupancy_map);
  if (!path.has_value()) {
    return std::unexpected(error::NO_VALID_PATH);
  }

  auto response = GetPath::Response{};
  /* More implementation code */
  return response;
}
```

- Here is the refactored core functionality of the generate path callback
- This function returns a type which can be used for **monadic error handling**
- If there is an error, the function can handle the error in a compile time checkable way
- The function that generates the path can now be passed in, making this function a **higher order function**
- This function is deterministic and has no side effects, so it is a **pure function**
- Let’s test this function

Testing the Refactored PathGenerator

```
TEST(GeneratePath, NoValidPath) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should be an error with the error::NO_VALID_PATH type
  EXPECT_EQ(response.error(), pathing::generate_path::error::NO_VALID_PATH);
}
```


PICKNIK

Testing the Refactored PathGenerator

```
TEST(GeneratePath, NoValidPath) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should be an error with the error::NO_VALID_PATH type
  EXPECT_EQ(response.error(), pathing::generate_path::error::NO_VALID_PATH);
}
```

- Testing the refactored functionality is trivial

Testing the Refactored PathGenerator

```
TEST(GeneratePath, NoValidPath) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should be an error with the error::NO_VALID_PATH type
  EXPECT_EQ(response.error(), pathing::generate_path::error::NO_VALID_PATH);
}
```

- Testing the refactored functionality is trivial
    - Create required parameters

Testing the Refactored PathGenerator

```
TEST(GeneratePath, NoValidPath) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should be an error with the error::NO_VALID_PATH type
  EXPECT_EQ(response.error(), pathing::generate_path::error::NO_VALID_PATH);
}
```

- Testing the refactored functionality is trivial
  - Create required parameters
  - Pass the parameters into the function under test

PICKNIK

Testing the Refactored PathGenerator

```
TEST(GeneratePath, NoValidPath) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should be an error with the error::NO_VALID_PATH type
  EXPECT_EQ(response.error(), pathing::generate_path::error::NO_VALID_PATH);
}
```

- Testing the refactored functionality is trivial
  - Create required parameters
  - Pass the parameters into the function under test
  - Check the return

PICKNIK

Testing the Refactored PathGenerator

```
TEST(GeneratePath, NoValidPath) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should be an error with the error::NO_VALID_PATH type
  EXPECT_EQ(response.error(), pathing::generate_path::error::NO_VALID_PATH);
}
```

- Testing the refactored functionality is trivial
  - Create required parameters
  - Pass the parameters into the function under test
  - Check the return
- All the functions that have been refactored so far can be tested this way

PICKNIK

Testing the Refactored PathGenerator

```
TEST(GeneratePath, NoValidPath) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should be an error with the error::NO_VALID_PATH type
  EXPECT_EQ(response.error(), pathing::generate_path::error::NO_VALID_PATH);
}
```

- Testing the refactored functionality is trivial
  - Create required parameters
  - Pass the parameters into the function under test
  - Check the return
- All the functions that have been refactored so far can be tested this way
- Everything can now be put together for the callback being executed by the generate path service

PICKNIK

Testing the Refactored PathGenerator

```
TEST(GeneratePath, NoValidPath) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should be an error with the error::NO_VALID_PATH type
  EXPECT_EQ(response.error(), pathing::generate_path::error::NO_VALID_PATH);
}
```

- Testing the refactored functionality is trivial
    - Create required parameters
    - Pass the parameters into the function under test
    - Check the return
- All the functions that have been refactored so far can be tested this way
- Everything can now be put together for the callback being executed by the generate path service
- **All of this has been done without invoking the ROS 2 API!**

PICKNIK

Putting it all together

```
[this](auto const request, auto response) {
    auto const print_error = [this](std::string_view error)
        -> std::expected<GetPath::Response, std::string> {...};

    auto const return_empty_response = []([[maybe_unused]] auto const)
        -> std::expected<GetPath::Response, std::string> {...};

    auto const stringify_error = [](auto const error) {...};

    *response = generate_path::generate_path(request, this->map_,
generate_global_path)
            .map_error(stringify_error)
            .or_else(print_error)
            .or_else(return_empty_response)
            .value();
  }
```

- The generate path callback function has been replaced by a lambda function

Putting it all together

```
[this](auto const request, auto response) {
    auto const print_error = [this](std::string_view error)
        -> std::expected<GetPath::Response, std::string> {...};

    auto const return_empty_response = []([[maybe_unused]] auto const)
        -> std::expected<GetPath::Response, std::string> {...};

    auto const stringify_error = [](auto const error) {...};

    *response = generate_path::generate_path(request, this->map_,
generate_global_path)
            .map_error(stringify_error)
            .or_else(print_error)
            .or_else(return_empty_response)
            .value();
  }
```

- The generate path callback function has been replaced by a lambda function
- If `generate_path` returns the expected value, it is directly assigned to `response`

PICKNIK

Putting it all together

```
[this](auto const request, auto response) {
    auto const print_error = [this](std::string_view error)
        -> std::expected<GetPath::Response, std::string> {...};

    auto const return_empty_response = []([[maybe_unused]] auto const)
        -> std::expected<GetPath::Response, std::string> {...};

    auto const stringify_error = [](auto const error) {...};

    *response = generate_path::generate_path(request, this->map_,
generate_global_path)
            .map_error(stringify_error)
            .or_else(print_error)
            .or_else(return_empty_response)
            .value();
  }
```

- The generate path callback function has been replaced by a lambda function
- If `generate_path` returns the expected value, it is directly assigned to `response`
- If `generate_path` returns an error, the error is handled by chaining functions together
  - This is the result of returning a monadic type and performing monadic error handling

PICKNIK

Putting it all together

```
[this](auto const request, auto response) {
    auto const print_error = [this](std::string_view error)
        -> std::expected<GetPath::Response, std::string> {...};

    auto const return_empty_response = []([[maybe_unused]] auto const)
        -> std::expected<GetPath::Response, std::string> {...};

    auto const stringify_error = [](auto const error) {...};

    *response = generate_path::generate_path(request, this->map_,
generate_global_path)
                .map_error(stringify_error)
                .or_else(print_error)
                .or_else(return_empty_response)
                .value();
  }
```

- The generate path callback function has been replaced by a lambda function
- If `generate_path` returns the expected value, it is directly assigned to `response`
- If `generate_path` returns an error, the error is handled by chaining functions together
  - This is the result of returning a monadic type and performing monadic error handling
- If needed, more functions can be added to manipulate the expected type or error type, increasing modularity

PICKNIK

Putting it all together

```
[this](auto const request, auto response) {
    auto const print_error = [this](std::string_view error)
        -> std::expected<GetPath::Response, std::string> {...};

    auto const return_empty_response = []([[maybe_unused]] auto const)
        -> std::expected<GetPath::Response, std::string> {...};

    auto const stringify_error = [](auto const error) {...};

    *response = generate_path::generate_path(request, this->map_,
generate_global_path)
            .map_error(stringify_error)
            .or_else(print_error)
            .or_else(return_empty_response)
            .value();
  }
```

- The generate path callback function has been replaced by a lambda function
- If `generate_path` returns the expected value, it is directly assigned to `response`
- If `generate_path` returns an error, the error is handled by chaining functions together
  - This is the result of returning a monadic type and performing monadic error handling
- If needed, more functions can be added to manipulate the expected type or error type, increasing modularity
- How can this lambda be tested?

PICKNIK

DI and Functional Programming

```
template <typename ServiceType>
using ServiceCallback =  std::function<void(
        std::shared_ptr<typename ServiceType::Request>const ,
        std::shared_ptr<typename ServiceType::Response>)>;

struct Manager {
  struct MiddlewareHandle {
    // Define map service callback type
    using SetMapCallback = ServiceCallback<SetMap>;

    // Define path generation service callback type
    using GeneratePathCallback = ServiceCallback<GetPath>;

    virtual ~MiddlewareHandle() = default;

    virtual void register_set_map_service(SetMapCallback callback) = 0;

    virtual void register_generate_path_service(GeneratePathCallback callback) = 0;

    virtual void log_error(std::string const& msg) = 0;

    virtual void log_info(std::string const& msg) = 0;
  };

  Manager(std::unique_ptr<MiddlewareHandle> mw);

 private:
  std::unique_ptr<MiddlewareHandle> mw_;

  Map<unsigned char> map_;
};
```

- With Dependency Injection (DI)!

DI and Functional Programming

```cpp
template <typename ServiceType>
using ServiceCallback =  std::function<void(
        std::shared_ptr<typename ServiceType::Request>const ,
        std::shared_ptr<typename ServiceType::Response>)>;

struct Manager {
  struct MiddlewareHandle {
    // Define map service callback type
    using SetMapCallback = ServiceCallback<SetMap>;

    // Define path generation service callback type
    using GeneratePathCallback = ServiceCallback<GetPath>;

    virtual ~MiddlewareHandle() = default;

    virtual void register_set_map_service(SetMapCallback callback) = 0;

    virtual void register_generate_path_service(GeneratePathCallback callback) = 0;

    virtual void log_error(std::string const& msg) = 0;

    virtual void log_info(std::string const& msg) = 0;
  };

  Manager(std::unique_ptr<MiddlewareHandle> mw);

 private:
  std::unique_ptr<MiddlewareHandle> mw_;

  Map<unsigned char> map_;
};
```

- With Dependency Injection (DI)!
  - DI is used to move or “inject” objects into another object

PICKNIK

DI and Functional Programming

```cpp
template <typename ServiceType>
using ServiceCallback =  std::function<void(
        std::shared_ptr<typename ServiceType::Request>const ,
        std::shared_ptr<typename ServiceType::Response>)>;

struct Manager {
  struct MiddlewareHandle {
    // Define map service callback type
    using SetMapCallback = ServiceCallback<SetMap>;

    // Define path generation service callback type
    using GeneratePathCallback = ServiceCallback<GetPath>;

    virtual ~MiddlewareHandle() = default;

    virtual void register_set_map_service(SetMapCallback callback) = 0;

    virtual void register_generate_path_service(GeneratePathCallback callback) = 0;

    virtual void log_error(std::string const& msg) = 0;

    virtual void log_info(std::string const& msg) = 0;
  };

  Manager(std::unique_ptr<MiddlewareHandle> mw);

 private:
  std::unique_ptr<MiddlewareHandle> mw_;

  Map<unsigned char> map_;
};
```

- With Dependency Injection (DI)!
  - DI is used to move or “inject” objects into another object
- There still needs to be mutable state, to keep track of the occupancy map between service calls, thus the `map_` member variable

DI and Functional Programming

```
template <typename ServiceType>
using ServiceCallback =  std::function<void(
        std::shared_ptr<typename ServiceType::Request>const ,
        std::shared_ptr<typename ServiceType::Response>)>;

struct Manager {
  struct MiddlewareHandle {
    // Define map service callback type
    using SetMapCallback = ServiceCallback<SetMap>;

    // Define path generation service callback type
    using GeneratePathCallback = ServiceCallback<GetPath>;

    virtual ~MiddlewareHandle() = default;

    virtual void register_set_map_service(SetMapCallback callback) = 0;

    virtual void register_generate_path_service(GeneratePathCallback callback) = 0;

    virtual void log_error(std::string const& msg) = 0;

    virtual void log_info(std::string const& msg) = 0;
  };

  Manager(std::unique_ptr<MiddlewareHandle> mw);

 private:
  std::unique_ptr<MiddlewareHandle> mw_;

  Map<unsigned char> map_;
};
```

- With Dependency Injection (DI)!
  - DI is used to move or “inject” objects into another object
- There still needs to be mutable state, to keep track of the occupancy map between service calls, thus the `map_` member variable
- For the `Manager` object, a `MiddlewareHandle` struct is defined that is the interface for the injected dependency
- This abstract interface can be used to implement each function using the ROS API

PICKNIK

DI and Functional Programming

```
template <typename ServiceType>
using ServiceCallback =  std::function<void(
        std::shared_ptr<typename ServiceType::Request>const ,
        std::shared_ptr<typename ServiceType::Response>)>;

struct Manager {
  struct MiddlewareHandle {
    // Define map service callback type
    using SetMapCallback = ServiceCallback<SetMap>;

    // Define path generation service callback type
    using GeneratePathCallback = ServiceCallback<GetPath>;

    virtual ~MiddlewareHandle() = default;

    virtual void register_set_map_service(SetMapCallback callback) = 0;

    virtual void register_generate_path_service(GeneratePathCallback callback) = 0;

    virtual void log_error(std::string const& msg) = 0;

    virtual void log_info(std::string const& msg) = 0;
  };

  Manager(std::unique_ptr<MiddlewareHandle> mw);

 private:
  std::unique_ptr<MiddlewareHandle> mw_;

  Map<unsigned char> map_;
};
```

- With Dependency Injection (DI)!
  - DI is used to move or "inject" objects into another object
- There still needs to be mutable state, to keep track of the occupancy map between service calls, thus the `map_` member variable
- For the `Manager` object, a `MiddlewareHandle` struct is defined that is the interface for the injected dependency
- This abstract interface can be used to implement each function using the ROS API
- The lambda function that is used for the generate path service can be captured via mocking and tested

PICKNIK

Testing with DI

```
struct PathManagerFixture : public testing::Test {
  PathManagerFixture() : mw_{std::make_unique<MockMiddlewareHandle>()} {
    // When the map callback is called, set the costmap
    ON_CALL(*mw_, register_set_map_service(testing::_))
        .WillByDefault([&](auto const& map_callback) {
          auto const map_request = make_occupancy_map();
          auto map_response = std::make_shared<SetMap::Response>();
          map_callback(map_request, map_response);
        });
    // Capture the path callback so it can be called later
    ON_CALL(*mw_, register_generate_path_service(testing::_))
        .WillByDefault(testing::SaveArg<0>(&path_callback_));
  }
  std::unique_ptr<MockMiddlewareHandle> mw_;
  pathing::Manager::MiddlewareHandle::GeneratePathCallback path_callback_;
};
TEST_F(PathManagerFixture, NoPath) {
  // GIVEN a path generator with a costmap
  auto const path_generator = pathing::Manager{std::move(mw_)};
  // WHEN the generate path service is called with an unreachable goal
  auto path_request = std::make_shared<GetPath::Request>();
  path_request->start.data = {2, 2};
  path_request->goal.data = {5, 5};
  auto path_response = std::make_shared<GetPath::Response>();
  path_callback_(path_request, path_response);
  // THEN the path generator should succeed
  EXPECT_EQ(path_response->code.code, example_srvs::msg::GetPathCodes::NO_VALID_PATH);
  auto const expected = pathing::Path{};
  // AND the path should be empty
  EXPECT_EQ(pathing::utilities::parseGeneratedPath(path_response->path), expected);
}
```

- Here is the code testing the generate path lambda function

PICKNIK

Testing with DI

```cpp
struct PathManagerFixture : public testing::Test {
  PathManagerFixture() : mw_{std::make_unique<MockMiddlewareHandle>()} {
    // When the map callback is called, set the costmap
    ON_CALL(*mw_, register_set_map_service(testing::_))
        .WillByDefault([&](auto const& map_callback) {
          auto const map_request = make_occupancy_map();
          auto map_response = std::make_shared<SetMap::Response>();
          map_callback(map_request, map_response);
        });
    // Capture the path callback so it can be called later
    ON_CALL(*mw_, register_generate_path_service(testing::_))
        .WillByDefault(testing::SaveArg<0>(&path_callback_));
  }
  std::unique_ptr<MockMiddlewareHandle> mw_;
  pathing::Manager::MiddlewareHandle::GeneratePathCallback path_callback_;
};
TEST_F(PathManagerFixture, NoPath) {
  // GIVEN a path generator with a costmap
  auto const path_generator = pathing::Manager{std::move(mw_)};
  // WHEN the generate path service is called with an unreachable goal
  auto path_request = std::make_shared<GetPath::Request>();
  path_request->start.data = {2, 2};
  path_request->goal.data = {5, 5};
  auto path_response = std::make_shared<GetPath::Response>();
  path_callback_(path_request, path_response);
  // THEN the path generator should succeed
  EXPECT_EQ(path_response->code.code, example_srvs::msg::GetPathCodes::NO_VALID_PATH);
  auto const expected = pathing::Path{};
  // AND the path should be empty
  EXPECT_EQ(pathing::utilities::parseGeneratedPath(path_response->path), expected);
}
```

- Here is the code testing the generate path lambda function
- This test fixture calls the callback function for the set occupancy map service when a mock function is executed

PICKNIK

Testing with DI

```
struct PathManagerFixture : public testing::Test {
  PathManagerFixture() : mw_{std::make_unique<MockMiddlewareHandle>()} {
    // When the map callback is called, set the costmap
    ON_CALL(*mw_, register_set_map_service(testing::_))
        .WillByDefault([&](auto const& map_callback) {
          auto const map_request = make_occupancy_map();
          auto map_response = std::make_shared<SetMap::Response>();
          map_callback(map_request, map_response);
        });
    // Capture the path callback so it can be called later
    ON_CALL(*mw_, register_generate_path_service(testing::_))
        .WillByDefault(testing::SaveArg<0>(&path_callback_));
  }
  std::unique_ptr<MockMiddlewareHandle> mw_;
  pathing::Manager::MiddlewareHandle::GeneratePathCallback path_callback_;
};
TEST_F(PathManagerFixture, NoPath) {
  // GIVEN a path generator with a costmap
  auto const path_generator = pathing::Manager{std::move(mw_)};
  // WHEN the generate path service is called with an unreachable goal
  auto path_request = std::make_shared<GetPath::Request>();
  path_request->start.data = {2, 2};
  path_request->goal.data = {5, 5};
  auto path_response = std::make_shared<GetPath::Response>();
  path_callback_(path_request, path_response);
  // THEN the path generator should succeed
  EXPECT_EQ(path_response->code.code, example_srvs::msg::GetPathCodes::NO_VALID_PATH);
  auto const expected = pathing::Path{};
  // AND the path should be empty
  EXPECT_EQ(pathing::utilities::parseGeneratedPath(path_response->path), expected);
}
```

- Here is the code testing the generate path lambda function
- This test fixture calls the callback function for the set occupancy map service when a mock function is executed
- This text fixture also captures the callback function for the generate path service so it can be executed later

PICKNIK

Testing with DI

```cpp
struct PathManagerFixture : public testing::Test {
  PathManagerFixture() : mw_{std::make_unique<MockMiddlewareHandle>()} {
    // When the map callback is called, set the costmap
    ON_CALL(*mw_, register_set_map_service(testing::_))
        .WillByDefault([&](auto const& map_callback) {
          auto const map_request = make_occupancy_map();
          auto map_response = std::make_shared<SetMap::Response>();
          map_callback(map_request, map_response);
        });
    // Capture the path callback so it can be called later
    ON_CALL(*mw_, register_generate_path_service(testing::_))
        .WillByDefault(testing::SaveArg<0>(&path_callback_));
  }
  std::unique_ptr<MockMiddlewareHandle> mw_;
  pathing::Manager::MiddlewareHandle::GeneratePathCallback path_callback_;
};
TEST_F(PathManagerFixture, NoPath) {
  // GIVEN a path generator with a costmap
  auto const path_generator = pathing::Manager{std::move(mw_)};
  // WHEN the generate path service is called with an unreachable goal
  auto path_request = std::make_shared<GetPath::Request>();
  path_request->start.data = {2, 2};
  path_request->goal.data = {5, 5};
  auto path_response = std::make_shared<GetPath::Response>();
  path_callback_(path_request, path_response);
  // THEN the path generator should succeed
  EXPECT_EQ(path_response->code.code, example_srvs::msg::GetPathCodes::NO_VALID_PATH);
  auto const expected = pathing::Path{};
  // AND the path should be empty
  EXPECT_EQ(pathing::utilities::parseGeneratedPath(path_response->path), expected);
}
```

- Here is the code testing the generate path lambda function
- This test fixture calls the callback function for the set occupancy map service when a mock function is executed
- This text fixture also captures the callback function for the generate path service so it can be executed later
- For this test the occupancy map has already been set via the test fixture

PICKNIK

Testing with DI

```cpp
struct PathManagerFixture : public testing::Test {
  PathManagerFixture() : mw_{std::make_unique<MockMiddlewareHandle>()} {
    // When the map callback is called, set the costmap
    ON_CALL(*mw_, register_set_map_service(testing::_))
        .WillByDefault([&](auto const& map_callback) {
          auto const map_request = make_occupancy_map();
          auto map_response = std::make_shared<SetMap::Response>();
          map_callback(map_request, map_response);
        });
    // Capture the path callback so it can be called later
    ON_CALL(*mw_, register_generate_path_service(testing::_))
        .WillByDefault(testing::SaveArg<0>(&path_callback_));
  }
  std::unique_ptr<MockMiddlewareHandle> mw_;
  pathing::Manager::MiddlewareHandle::GeneratePathCallback path_callback_;
};
TEST_F(PathManagerFixture, NoPath) {
  // GIVEN a path generator with a costmap
  auto const path_generator = pathing::Manager{std::move(mw_)};
  // WHEN the generate path service is called with an unreachable goal
  auto path_request = std::make_shared<GetPath::Request>();
  path_request->start.data = {2, 2};
  path_request->goal.data = {5, 5};
  auto path_response = std::make_shared<GetPath::Response>();
  path_callback_(path_request, path_response);
  // THEN the path generator should succeed
  EXPECT_EQ(path_response->code.code, example_srvs::msg::GetPathCodes::NO_VALID_PATH);
  auto const expected = pathing::Path{};
  // AND the path should be empty
  EXPECT_EQ(pathing::utilities::parseGeneratedPath(path_response->path), expected);
}
```

- Here is the code testing the generate path lambda function
- This test fixture calls the callback function for the set occupancy map service when a mock function is executed
- This text fixture also captures the callback function for the generate path service so it can be executed later
- For this test the occupancy map has already been set via the test fixture
- The generate path callback can now be tested by executing the callback function directly

PICKNIK

Testing with DI

```
struct PathManagerFixture : public testing::Test {
  PathManagerFixture() : mw_{std::make_unique<MockMiddlewareHandle>()} {
    // When the map callback is called, set the costmap
    ON_CALL(*mw_, register_set_map_service(testing::_))
        .WillByDefault([&](auto const& map_callback) {
          auto const map_request = make_occupancy_map();
          auto map_response = std::make_shared<SetMap::Response>();
          map_callback(map_request, map_response);
        });
    // Capture the path callback so it can be called later
    ON_CALL(*mw_, register_generate_path_service(testing::_))
        .WillByDefault(testing::SaveArg<0>(&path_callback_));
  }
  std::unique_ptr<MockMiddlewareHandle> mw_;
  pathing::Manager::MiddlewareHandle::GeneratePathCallback path_callback_;
};
TEST_F(PathManagerFixture, NoPath) {
  // GIVEN a path generator with a costmap
  auto const path_generator = pathing::Manager{std::move(mw_)};
  // WHEN the generate path service is called with an unreachable goal
  auto path_request = std::make_shared<GetPath::Request>();
  path_request->start.data = {2, 2};
  path_request->goal.data = {5, 5};
  auto path_response = std::make_shared<GetPath::Response>();
  path_callback_(path_request, path_response);
  // THEN the path generator should succeed
  EXPECT_EQ(path_response->code.code, example_srvs::msg::GetPathCodes::NO_VALID_PATH);
  auto const expected = pathing::Path{};
  // AND the path should be empty
  EXPECT_EQ(pathing::utilities::parseGeneratedPath(path_response->path), expected);
}
```

- Here is the code testing the generate path lambda function
- This test fixture calls the callback function for the set occupancy map service when a mock function is executed
- This text fixture also captures the callback function for the generate path service so it can be executed later
- For this test the occupancy map has already been set via the test fixture
- The generate path callback can now be tested by executing the callback function directly
- **There was no invocation of the middleware using DI and all code is testable without invoking the ROS 2 API!**

PICKNIK

Conclusion

Conclusion

- The refactored tests are deterministic - they cannot be flaky

Conclusion

- The refactored tests are deterministic - they cannot be flaky
- Break down code into discrete components that can be tested

PICKNIK

Conclusion

- The refactored tests are deterministic - they cannot be flaky
- Break down code into discrete components that can be tested
- Prioritize using pure functions - easier to test and reason about

Conclusion

- The refactored tests are deterministic - they cannot be flaky
- Break down code into discrete components that can be tested
- Prioritize using pure functions - easier to test and reason about
- Using higher order functions increased the modularity of the code, in this case allowing for different path generating algorithms to be used

PICKNIK

Conclusion

- The refactored tests are deterministic - they cannot be flaky
- Break down code into discrete components that can be tested
- Prioritize using pure functions - easier to test and reason about
- Using higher order functions increased the modularity of the code, in this case allowing for different path generating algorithms to be used
- Monadic error handling led to easier error checking

Conclusion

- The refactored tests are deterministic - they cannot be flaky
- Break down code into discrete components that can be tested
- Prioritize using pure functions - easier to test and reason about
- Using higher order functions increased the modularity of the code, in this case allowing for different path generating algorithms to be used
- Monadic error handling led to easier error checking
- Refactoring `PathGenerator` using DI in conjunction with the functional programming paradigm led to code that has 100% coverage

Thanks to:

- Mariyum Gill
- Griswald Brooks
- Davide Faconti
- Kyle Kirves
- Tyler Weaver
- Chris Thrasher
- Brian Cairl
- Everyone else at PickNik

Bilal Gill

Leveraging a Functional Approach for More Testable and Maintainable ROS 2 Code

Thank you!

All code and the full presentation are available at:

PICKNIK